The PGAS model & Introduction to UPC

Dr Michèle Weiland
Project Manager, EPCC
The University of Edinburgh
Outline of talk

• HPC, parallel architectures & the motivation behind PGAS

• The PGAS programming model

• Introduction to UPC
  ‣ basic concept
  ‣ data distribution & blocking factors
  ‣ synchronisation & work sharing
  ‣ pointers, dynamic memory allocation & collectives
What is HPC?

• High performance computing = parallel computing
  ‣ distributing computation over many CPUs

• Performance is the key
  ‣ aim is to make codes run faster!
  ‣ not to possible to simply use faster CPUs (heat, power, physical limitations)
What is HPC?

• Maximise parallel speed-up $S(P)$ on $P$ processors

\[
S(P) = \frac{T(1)}{T(P)}
\]

➢ parallel *algorithms* to solve science
➢ parallel *codes* that implement algorithms
➢ parallel *machines* to run codes
Parallel architectures

- Shared memory
  - each processor has access to a global memory store
  - communications via memory reads/writes
Parallel architectures

- Distributed memory
  - each processor has its own memory and runs a copy of the OS
  - communication via the interconnect

In recent years, these single processors have become multi-core chips or heterogeneous nodes with accelerators.
Parallel programming paradigms

• Data parallelism
  ‣ divide data into subsets, process all subsets in the same way

• Task parallelism
  ‣ divide problem into independent tasks and process tasks in parallel

⇒ divide a large problem up into smaller problems!
Challenges facing HPC going forward

- Systems have many tens of thousands of cores
  - will go up to millions before end of decade
- Programmability of heterogeneous systems
- Power/energy usage

- We need
  - better algorithms
  - software designed to take advantage of architecture
  - improved parallel programming models
PGAS programming model - why?

• Parallel programming is hard because mainstream languages were designed for serial programming

• No support for parallelism in the languages - specialist libraries are required

• High level of complexity does not encourage well written and properly designed software...

➡ MPI (Message Passing Interface) library and OpenMP API are currently the most widely used approaches in parallel applications

➡ Accelerators have added CUDA and OpenACC to the mix
PGAS

- **Partitioned Global Address Space**
  - logically partitioned
  - local portions for each process
PGAS vs MPI

- multi-threaded control
- global name space
- single-sided communication
- explicit parallel syntax

- multi-threaded control
- private name space
- mostly two-sided communication
- explicit communication
PGAS languages

- UPC
- Fortran Coarrays
- Chapel
- Fortress
- X10

Language extensions
New developments
UPC

• Unified Parallel C

• Parallel extension to ISO C99
  ‣ with global shared address space
  ‣ and explicit parallelism & synchronisation

• Both commercial and open-source compilers available
  ‣ LNBL & UC Berkley: http://upc.lbl.gov
  ‣ GNU UPC: http://www.gccupc.org
UPC and the world of PGAS

• PGAS is a programming model

• UPC is one implementation of the model
  ‣ there are many other implementations
  ‣ all implementations are different, but fundamental concept remains the same!
Private vs shared data

• concept of two memory spaces: **private** and **shared**

• **private** variables are declared as normal C variables
  ‣ multiple instances will exist

```c
int x; // private variable
```

• **shared** variables are declared with shared qualifier
  ‣ only allocated once, accessible by all threads

```c
shared int y; // shared variable
```
The UPC model
UPC basics

- UPC threads operate independently in SPMD fashion

- Two variables for querying environment:
  - THREADS: holds total number of threads
  - MYTHREAD: stores thread index (runs from 0 to THREADS-1)

```c
#include <upc.h>
#include <stdio.h>

void main()
{
  printf("Thread %d of %d says: Hello!", MYTHREAD, THREADS);
}
```
Data distribution

- if a shared variable is scalar, space is allocated on thread 0 only

```c
int x;
shared int y;
```
Data distribution

- if a shared variable is an array, space is by default allocated across shared memory space in cyclic fashion

```c
int x;
shared int y[8];
```

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

private memory space

shared memory space
Data distribution

- if the number of elements in the shared array does not divide by the number of threads, the distribution will be uneven

```c
int x;
shared int y[9];
```
Data distribution

- change the default data layout by adding a "blocking factor" to shared arrays

```
int x;
shared [2] int y[8];
```

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

private memory space

shared memory space
2D array decomposition

```
shared [6] int a[8][8];
```

important to think about how blocking factor can impact data layout!

example on 4 threads
Blocking factor

• should be used if default distribution is not suitable

• four different cases:

  ‣ **shared** [n]: defines a block size of n elements

  ‣ **shared** [0]: all elements are given affinity to thread 0

  ‣ **shared** [*]: when possible, data is stored in contiguous blocks

  ‣ **shared** [:]: equivalent to **shared** [0]
Static vs dynamic compilation

- number of UPC threads can be specified at *compile time* (static) or at *runtime* (dynamic)

- Advantages
  - dynamic: program can be executed using any number of threads
  - static: easier to distribute data based on THREADS

- Disadvantages
  - dynamic: not always possible to achieve best possible distribution
  - static: program needs to be executed with number of threads specified at compile time
Static vs dynamic compilation

“An array declaration is illegal if THREADS is specified at runtime and the number of elements to allocate at each thread depends on THREADS.”

- `shared int x[4*THREADS];`
  - legal for static and dynamic environment
- `shared[] int x[8];`
- `shared int x[8];`
- `shared[] int x[THREADS];`
  - illegal for dynamic environment
- `shared int x[10+THREADS];`
Side-effects of shared data

Holding data in shared memory space has implications

1. the lifetime of the shared data needs to extend beyond the scope in which it was defined
   - storage duration

2. the shared data needs to be kept up-to-date
   - synchronisation
Storage duration of shared objects

Shared objects cannot have automatic storage duration

- any variable inside a function!

Why?

- SPMD model means a shared variable may be accessed outside the lifetime of the function!

Shared variables must either have file scope or be declared with static keyword
Synchronisation

• SPMD model means threads operate independently

• Synchronisation vital to ensure all threads reach same point in execution

  ▸ necessary for memory and data consistency

  ▸ only read data that is up-to-date, only overwrite data that is no longer needed

• UPC uses barriers for synchronisation

  ▸ most commonly used: upc_barrier
Work sharing with upc_forall

- 4th parameter defines affinity to thread

```
upc_forall (expression; expression; expression; affinity)
```

- **Condition**: iterations of upc_forall must be independent!

  if “pointer to shared”:
  
  object pointed to has affinity to MYTHREAD

  if integer expression:
  
  affinity%THREADS == MYTHREAD
Example: maximum of an array

```c
#define max(a,b) (((a)>(b)) ? (a) : (b))

shared int maximum[THREADS];
shared int globalMax = 0;
shared int a[THREADS*10];

void main(int argc, char **argv) {
    ... // initialise array a

    upc_barrier;
    upc_forall(int i=0; i<THREADS*10; i++; i){
        maximum[MYTHREAD] = max(maximum[MYTHREAD], a[i]);
    }
    upc_barrier;
    if (MYTHREAD == 0){
        for (int thread=0; thread<THREADS; thread++){
            globalMax = max(globalMax, maximum[thread]);
        }
    }
    upc_barrier;
    ...
}
```

- Here: shared variables have file scope!
- Ensure all threads found local maximum.
- Make sure globalMax is found before being used!
Example: vector addition (1/4)

- three vectors with default distribution - modulo operation identifies local elements

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
    int i;

    for(i=0; i<N; i++)
        if (MYTHREAD == i%THREADS)
            v1plusv2[i] = v1[i] + v2[i];
}
```

if distribution changes, this code will fail to identify local elements - however it will still produce the correct result!
Example: vector addition (2/4)

- alternative implementation would iterate in steps of THREADS and eliminate the need for the modulo operation

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
  int i;

  for(i=MYTHREAD; i<N; i+=THREADS)
    v1plusv2[i] = v1[i] + v2[i];
}
```

if distribution changes, this code will fail to identify local elements - however it will still produce the correct result!
Example: vector addition (3/4)

- implementation using upc_forall, taking advantage of the affinity parameter

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
    int i;
    upc_forall(i=0; i<N; i++; i) {
        v1plusv2[i] = v1[i] + v2[i];
    }
    i is short for i%THREADS=MYTHREAD
}
```
Example: vector addition (4/4)

- implementation using upc_forall, taking advantage of the affinity parameter

```c
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
  int i;
  upc_forall(i=0; i<N; i++; &v1plusv2[i])
    v1plusv2[i] = v1[i] + v2[i],
}
```

identifies local elements correctly regardless of distribution
Working sharing & performance

- perform as much computation as possible on local data

  - remote memory operations are expensive!

```c
#include <upc.h>
shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void) {
    int i, j;

    upc_forall( i = 0 ; i < THREADS ; i++ ; &c[i]) {
        c[i] = 0;
        for ( j= 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```

Ensure matrix A is distributed by row.
UPC pointers

1. private to private: `int *p1;`
   - Standard C pointer

2. private to shared: `shared int *p2;`
   - Private pointer into the shared memory space

3. shared to private: `int *shared p3;`
   - Shared pointer into the private memory space - not recommended!

4. shared to shared: `shared int *shared p4;`
   - Shared pointer into the shared memory space

---

Diagram: [Diagram showing memory spaces and pointer types]
Dynamic memory allocation

- in private memory space, usual C functions apply
- in shared space, UPC offers three different functions
  - upc_alloc: allocate local shared spaces
  - upc_global_alloc: allocate multiple global spaces
  - upc_all_alloc: allocate a global shared memory space collectively
- upc_free used to deallocate shared memory
UPC collectives

- requires upc_collective.h header file

- implemented by most compilers, but performance not necessarily optimised

- two types of collectives
  - relocalisation: upc_all_broadcast, upc_all_scatter, upc_all_gather, ...
  - computational: upc_all_reduceT, upc_all_sort, ...

- calls to these functions must be performed by all threads
Broadcast

```c
#include <upc_collective.h>

shared [] int A[2];
shared [2] int B[N][2];

upc_all_broadcast(B, A, 2*sizeof(int), UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```
PGAS programming model

• global view paradigm

• explicit support for parallelism

• compiler can help programmer with performance, scalability, programmability
  
  ‣ we are still quite a long way from this goal

• potential reduction in memory footprint = reduction in energy consumption
References & further reading

• UPC Language Specification (Version 1.2) on the Berkley Unified Parallel C project homepage:
  
  http://upc.gwu.edu/docs/upc_specs_1.2.pdf

• GNU Unified Parallel C toolset: http://www.gccupc.org

• Tarek El-Ghazawi et al. “UPC: Distributed Shared Memory Programming”. Available through the Wiley Online Library.